Table of Contents

1. Introduction
2. Test Suites
3. Acceptance tests
4. Inspection Tests
5. Bibliography
6. **Introduction**

“The Haunted Building is a single-user role playing game. Before start the game, player will choose character to play first. In the building, every floor has more than 2 elevator that can go upstairs or downstairs. Player needs to escape the building as soon as possible by solving riddles and performing quests.” – Project Overview from The Haunted Building Final Report

In this unit testing and inspection report we developed various testing suites and test cases for important functions of the program. We used notation **BB#** and **WB#** to specify a black box test or white box test respectively. Some of our code included form code where input was attained via textboxes. Other C# functions were tested by hard coding values, compiling, and running the program and recording the result. Furthermore, acceptance tests were done by playing the game and looking for a specific functionality. For inspection, we visually reviewed code for readability, style, comments, consistency, spacing, bugs, unused code. Anything considered faulty or code that does not meet a certain requirement was noted appropriately.

1. **Test Suites**

**Login Form**

Author: Nooshin Mojab   
Testers: Nooshin Mojab, Justo Diaz

This is a windows form with two textboxes where the user inputs their login credentials. It must be displayed before the game starts as it is required for players to have accounts. Input is various ASCII strings and we record the result entering invalid strings. Accounts are stored in a database.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Case | txtUser | txtPass | Expected | Result |
| BB1 | Empty String | Valid String | Display a message "enter your Username" | Display two messages "enter your username" and "invalid username/password" |
| BB2 | Valid String | Empty String | Display a message "enter your Password" | Display two messages "enter your Password" and "invalid username/password" |
| BB3 | Empty String | Empty String | Display a message "enter your Username" | Display two messages "enter your username" and "invalid username/password" |
| BB4 | Invalid Username | Invalid Password | Display a message "invalid Username /Password" | Display a message "invalid Username /Password" |
| WB1 | Input valid strings. Statements executing SQLConnection with connectionInfo. | | Cnn.Open() should not raise exception. | No SqlException raised. |
| WB2 | Input valid strings, but with credentials not in database | | If(Dr.Read()) should be false, so we run else statement | Else statement code ran, displayed textbox. |

**Create New Account Form**

Author: Nooshin Mojab   
Testers: Nooshin Mojab, Justo Diaz

In this form , the user choose from the previous form to create a new account. They must enter a supported user name string and a valid password. If valid, the account is created in the database, otherwise appropriate error messages are displayed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Case | txtUser | txtPass | Expected | Result |
| BB1 | Empty String | Empty String | Display a message "Please enter username and password", do not let user play a game | Create an account and user is able to play a game |
| BB2 | txtUser.length>50 | 0<txtUser.length<=50 | Display an error message "The length of username must be less than 50 characters", do not let user play a game | Create an account and user is able to play a game |
| BB3 | 0<txtUser.length<=50 | txtUser.length>50 | Display an error message "The length of password must be less than 50 characters", do not let user play a game | Create an account and user is able to play a game |
| BB4 | Sql hacking code | Sql hacking code | Will be recognized as a valid username or password and does not have any bad effect on database | Will be recognized as a valid username or password and does not have any bad effect on database |
| BB5 | 0<txtUser.length<=50 | 0<txtUser.length<=50 | Next form will be shown while "continue save game" is disabled and "play new game" is enabled | Next form will be shown while "continue save game" is disabled and "play new game" is enabled |
| WB1 | Input valid strings. Statements executing SQLConnection with connectionInfo. | | Cnn.Open() should not raise exception. | No SqlException raised. |
| WB2 | Input valid strings. Change fmNewAcc\_Load() values of assignment. | | TaxIdex and Center to Screen should change | Changes too affect. |

**Start Game**

Author: Justo Diaz  
Testers: Nooshin Mojab, Justo Diaz

This function is located in HauntedBuilding.cs and it’s called from the start button click handler in fmPlayGame.cs. It’s the initial function that sets everything up based on the parameters passed to it. The function itself calls subroutines to setup player, floor, and elevators. An error check function is called before that error checks the parameters. If the error checks find an invalid input, it should stop the game. We tested invalid inputs for each parameter, while keeping other parameters valid.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Input | TC1 | Expected | Result | TC2 | Expected | Result |
| PlayerName | Empty String | Close the program | Still playing | Length of PlayerName>50 | Display an error message, close the program | Still playing |
| FloorNumber | Negative value | Display an message, close the program | Raise exception, program crash | Out of bound value depends on difficulty | Display an message, close the program | Raise exception, program crash |
| Coord | Out of bound value | Display an message, close the program | Raise exception, program crash |  |  |  |
| Difficulty | Negative value | Display an message, close the program | Go through infinite loop, freeze the program | Value greater than 3 | Display an message, close the program | Go through infinite loop, freeze the program |
| CaseHint | Empty String | Continue the game | Continue the game | Random string | Continue the game | Continue the game |

**Generate Random Sequence**

Author: Salvador Ariza  
Testers: Nooshin Mojab, Justo Diaz

This function is called in setupElevators(). It creates a random correct sequence of floors that must be traversed in order, starting from floor 10. The function is given a parameter of the number of floors and it should return a valid sequence.

|  |  |  |  |
| --- | --- | --- | --- |
| Case | Number of floors | Expected | Result |
| BB1 | <0 | Display an error message, close the program | Overflow exception, program crash |
| BB2 | 0 | Display an error message, close the program | Index out of range exception, program crash |
| BB3 | 1 | Continue the game | Continue for this function, fail for the rest (out of bound) |
| BB4 | >1 | Continue the game | Continue the game |

**Add Monster**

Author: Evan Currier  
Testers: Nooshin Mojab, Justo Diaz

This function is called in the Floor constructor, places a monster on the floor at a random location. The number of monsters placed depends on the difficulty. We tested the only parameter to the function, current, which must always equal 4. Anything else is invalid.

|  |  |  |  |
| --- | --- | --- | --- |
| Case | Current | Expected | Result |
| BB1 | <0 | Display an error message, close the program | Out of bound exception, program crash |
| BB2 | 0-3 | Display an error message, close the program | Continue the game and Monsters will be displayed as Items |
| BB3 | 4 | We have the Monster display on Screen | We have the Monster display on Screen |
| BB4 | >4 | Display an error message, close the program | Out of bound exception, program crash |

**Save Game**

Author: Nooshin Mojab  
Testers: Nooshin Mojab, Justo Diaz

The function is a click handler for the save button. We tested what occurs when the player loses due to max scare meter or timed out. The button should be disabled, but this is not the case.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Case | ScareMeter | TimeRemain | Expected | Result |
| BB1 | Reaches the max value | >0 | Disable Save button | Save button is not disabled, User can click on save button but it does not save the player's game |
| BB2 | Does not reach the max value | =0 | Disable Save button | Save button is not disabled, User can click on save button but it does not save the player's game |
| WB1 | Valid inputs, but have an invalid database file so that Cnn.Open() raises an exception. | | The game should continue, but notify the player that their game was not saved | Detailed SQL specific message, does not help the user. |

Use cases were tested as specified in “The Haunted Building” final report. This is more of a client acceptance tests to see if we delivered based on the use case diagram. All tests were done with the executable as a black box.

|  |  |  |  |
| --- | --- | --- | --- |
| Use Case | Input | Expected | Result |
| Launch application | Double click on application icon | Provide new users an option of creating account and existing user logging in to the game  After user log in the game, a pop up window will provide a user an option of choosing difficulty | Pass |
| Collect Coin | Player moves close to a spot where there is a coin. | Coin shines brightly to imply that there is money to be collected. Player moves to the spot and does a left click to collect the coin. A “Ding” sound plays out implying the coin has been successfully collected and a number “+5/10/15” would be displayed to the screen showing the amount of coins collected. | Failed, hourglasses were added instead. |
| Control Mouse/Keyboard | Player uses control unit (mouse, keyboard) to make a move. | Haunted Building updates the screen of where the Player is. | Mouse failed. Keyboard passed. |
| Solve Puzzles | Player moves close to a puzzle. Haunted Building shows a star implying a puzzle is there. Player right click the mouse to enter the puzzle. | Haunted Building gives puzzles with different difficulty levels for Player to choose, with different cost of coins. 1) Player successfully solve the puzzle. A hint is given and stored in the backpack  2) Player fails to solve the puzzle. No hint given to help leaving the haunted building | Failed |
| Take Elevators | Player moves to a door where an elevator is.  Go to elevator spot. Key press X and C to take elevator. | Go to next correct floor in sequence.  Initially a player is on top of the building | Pass |
| Make Moves | Moving on floor using key press W,A,S,D  When mouse is moved | W goes up, A goes left, S goes down, D goes right | Keyboard input passed, mouse input fail |
| Picking up the items | Pick up the items using key press E | The item get removed from the floor, and the player will have that item in his/her inventory | Pass |
| Take Hints | Player goes near a hint. Player takes hint by clicking it or press space button on keyboard | Haunted Building shows a silver light implying there is a hint to take. | Pass, take item with hint with ‘e’ key press. |
| Search Backpack | Move mouse/Keyboard close to the Backpack icon in the game window | Hints/Coins/Query items collected and stored in the backpack are displayed for the Player to make decisions | Failed |
| Leave Building | Player moves close to the elevator that sends him to the first floor. Haunted Building changes the elevator door color to golden. Player takes the elevator by left clicking the mouse or press space key on the keyboard | Haunted Building sends the Player to the first floor and opens the gate of the Haunted Building. An animation is played after the Player leaves the Haunted Building | Failed |
| Load/Save Game | Player presses upper right of the Haunted Building game window. | Haunted Building shows a dropdown menu given choices to load/save game. The Player saves the game by going to the “Save” item. The Player loads the game by going to the “Load” item | Save feature pass. Load feature failed. |

1. **Acceptance Tests**

The following is a list of the functional requirements from “The Haunted Building” Report. We ran tests using the executable as a black box with the inputs specified in the requirement and checked off whether the requirement was achieved. What the output should be is also specified in the requirement.

1. The user must be able to log in to the game by using the username and password, or the user can create an account with a new username and a password. **Pass.**
2. The game can save and track the previous level the user was in, and when the user is logged in again the game will continue from where it was left. (i.e. The level, time and the hints found should be tracked.) **Pass.**
3. The system shall be able to change to an offline mode. For the next time, the user tries to connect to the server again then the status (the user in) will be uploaded to the server (sync). **Fail.**
4. In the game, every room must have a splash screen or introduction before playing the actual game, so that the user knows what to press and what they need to do to find the hint. (i.e. a graphic sticker showing what they need to find in the room). **Pass.**
5. The user can use the mouse to “pick up” the hints/tools in the room and “drag them into” the pocket slots on the screen. (i.e. a key on the floor which is used to open a safe box that can be dragged to the pocket slot by the mouse). **Fail. Note: Can only pickup.**
6. The game should be able to give a board view of the building, so the users will know which floor they are on. **Pass.**
7. Games will be programmed as applets, as the implementation idea is used in the GUI to launch the game. **Fail.**
8. SWING/AWT systems are available, however, the game player will concentrate on swing as it has some nicer features, and have all game modules using strictly the one we pick, so everything is uniform. **Fail.**
9. The game should allow the user to use the top-10 feature. So the user can see the people/group (username) who finished the game. (Will be a text file separate from the code itself and it will be read into the module) **Fail.**
10. The game should allow the user to play in multi mode. So they will have a chat window showing the online group of people, in addition, they can chat in a chat window. **Fail.**
11. There must be a way to turn music [on/off]. **Fail.**
12. A HELP document build in, so that the user can pause the game, and view the help option. (Will be a text file separate from the code itself, and it will read into the module). **Pass. Note: No document, but help text pauses game.**
13. The system should be able to send out the notification. (i.e. update information, server maintenance). **Fail.**
14. There must be a check for the updates to get the current information or the new feature from the server. (When the server sends out the notification for the new feature updates). **Fail.**

Additional acceptance tests on part 17 of the final report.

**Look and Feel Requirements**

**Appearance Requirements**

The product shall be attractive to young adults between the age of 18 – 30 years old.

The product will have simple operation.

The colors used in the product will be dark and grayish colors, giving it a scary and a gloomy look.

**Result:**  Scary and gloomy look achieved to a certain extent. Displaying .gif images adds to the visual experience.

**Style Requirements**

The style of the product will have background of a ghost-house.

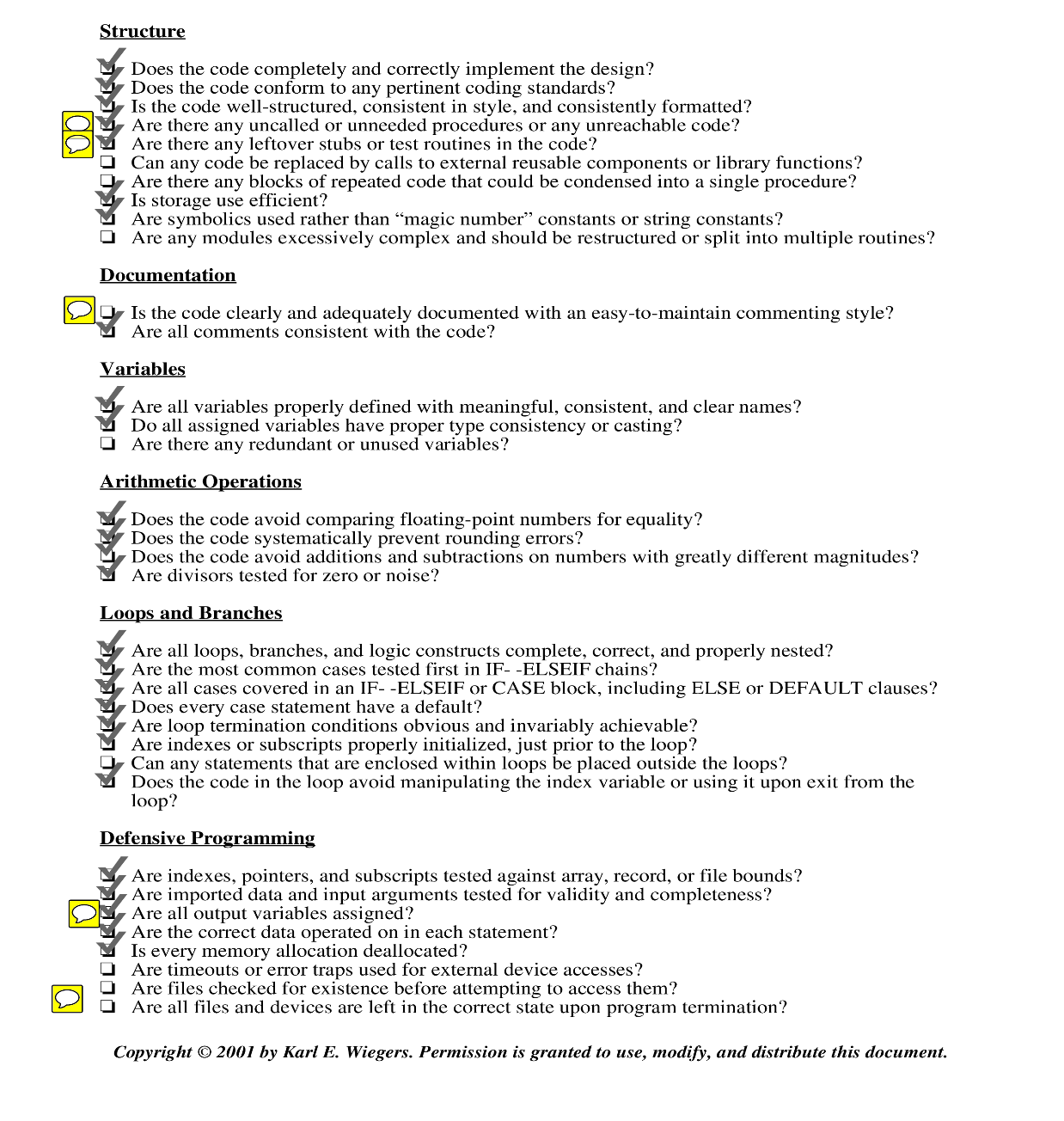
The product will have wind or ghost sounds. If the player has wrong answer for quiz/puzzle, there would be a screaming sound and ghost image shall show up.

Every floor will have different ghost-theme, which gives the player variety.

**Result:** Failed.

1. **Inspection Tests**

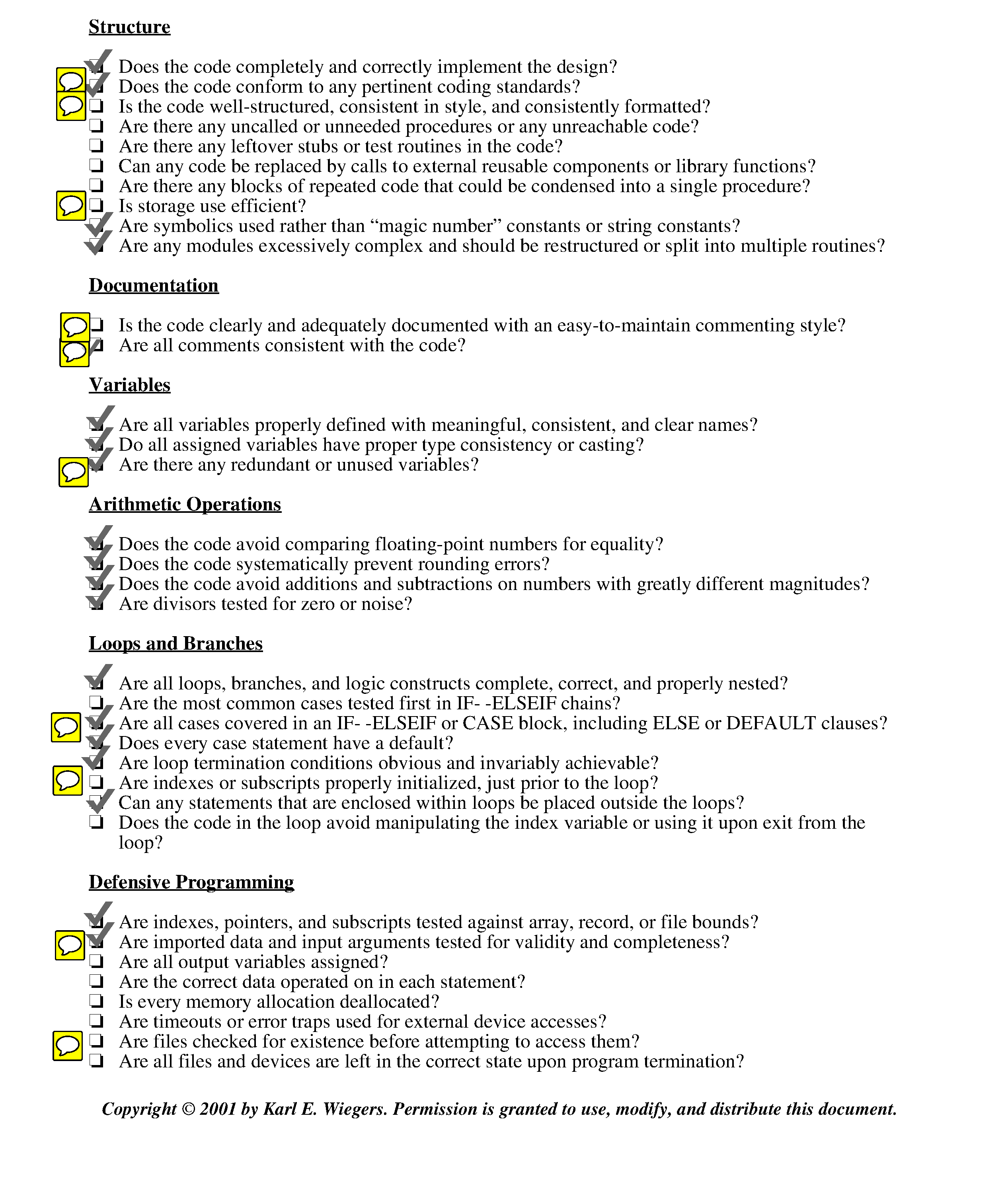
With the aid of a generic code review Inspection list, which we will reference and give credit to on our bibliography section, we have been able to generate the following results after close inspection of our following pieces of code: Player Object Class, setupElevators method , Monster Object Class, Saved Game method, and. The following figures below, which are ordered in sequence to the latter list specified, each display the content of what was tested followed by a check mark if the particular content passed or failed inspection. Extra yellow highlighted thought bubbles can be seen and denote extra comments in regards to the tested content. As a result, a list of those particular comments will follow each figure, in order of each thought bubble given.

Figure 1 Player Object Class Inspection Results

**Further Comments:**

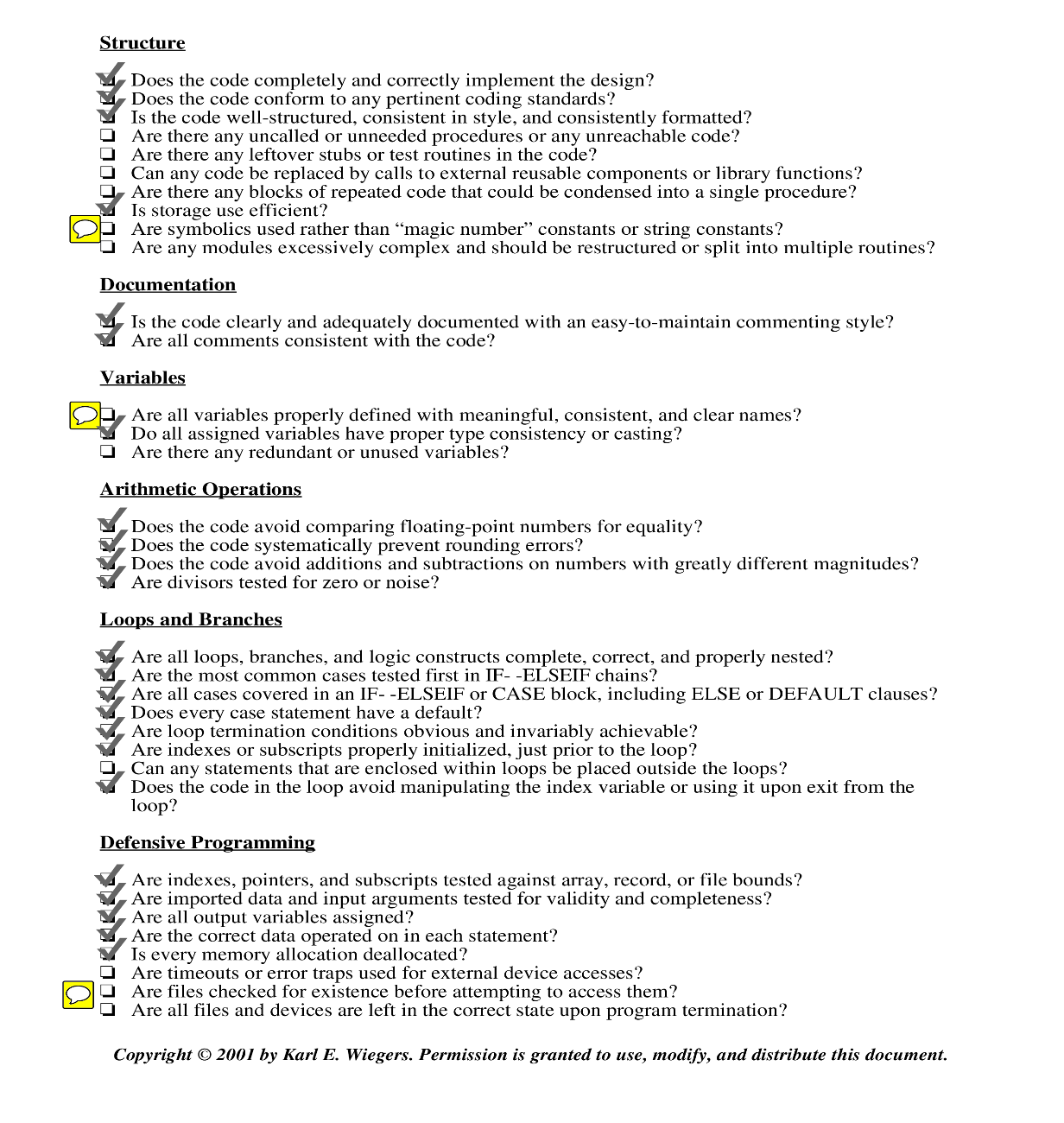
* A following method *enterElevator()* is not used at all and could be remnants of prior game design decisions.
* There are leftover methods as can be seen by the prior comment. *enterElevator()* is most definitely not used and not needed.
* Sufficient documentation is not given about the code. One-line comments are given about functions but are not sufficient detail to understand the logic.
* Aside from strings, other output variables are assigned a value prior to being returned
* Files are not checked for existence since our program does not require the use of files.

Figure 2 setupElevators Inspection Results



**Further Comments:**

* Code is well-nested and has consistent a naming convention.
* Variable names are not consistently formatted. Some of the naming conventions use underscores, while others use camel-case.
* currLoc, prevLoc, and nextLoc are initialized every time throughout the for-loop. They could be defined once in the beginning of the function and instead reassigned in the for-loop.
* There is barely enough documentation.
* There is probably a bit too much documentation over the for-loop
* An extra pair of local variables x2 and y2 are given but mildly used. Also need not be used.
* Inside the second for loop, the boundary cases of the loop are tested (e.g. when int i = 0, int i = anything else in between, int i = n-1)
* Variables are initialized inside the for-loop as opposed to prior to the for-loop.
* Imported data is correct assuming the components it relies on are valid as well
* Files are not used in this component so it is not applicable.

Figure 3 Monster Object Class Inspection Results

**Further Comments:**

* Globals and constants are used to figure out how many monsters to add. As can be seen below.

else if (i == (int)iName.MONSTER) {

while (monsterCount < **Globals.**NUM\_MONSTERS) {

addMonster(**Constants.**taken);

monsterCount++;

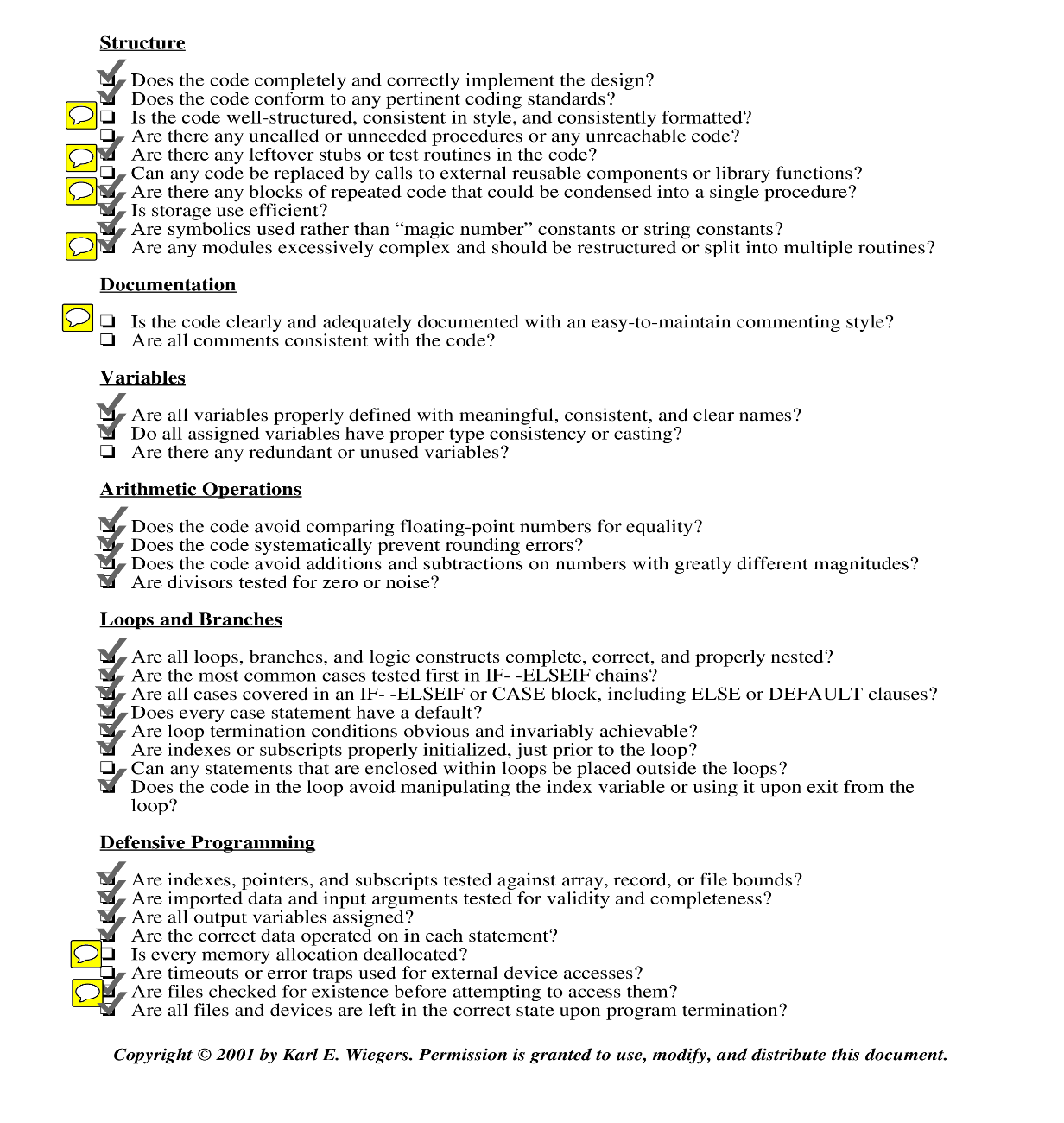
}

continue;

}

* Monster\_Count is a vague name. A more proper name such as Monster\_Counter could have been more appropriate, for instance.
* As for all the other components so far, files are not needed for our program and thus it is not applicable

Figure 4 Saved Game Inspection Results



**Further Comments:**

* This particular code has a lot of variables being assigned and entered as parameter. While it is not wrong, it is indeed messy.
* There is leftover commented-out code at the end of the function.
* A function built around handling the assignation of the parameters would definitely benefit the code by reducing a lot of clutter in the *saveGame()* function.
* Additional modules, such as a function that is built around assigning parameter values would aid in restructuring the code and splitting it into more sophisticated subroutines.
* While there isn't much documentation, the function is not overly complex and does not warrant in depth comments. However, some documentation to at least cover the function as a whole would be helpful. Most comments in this function are just previous bits of code that haven’t been deleted.
* Deallocation is automatically handled by c#.
* A try-exception block is used to make sure that a file can be opened correctly.

1. **Bibliography**

The Haunted Building Final Report by Qian Wang, Ze Li, Siddharth Sinha, Bora Park.

Test Suite Example  
<http://www.tutorialspoint.com/software_testing_dictionary/test_suite.htm>

Adapted Generic Code Review Inspection List

<https://www.liberty.edu/media/1414/%5B6401%5Dcode_review_checklist.pdf>